

16. Creating and modifying *e* variables

This clause describes how to create and assign values to *e* variables.

16.1 About *e* variables

An *e* variable is a named data object of a declared type. *e* variables are declared and manipulated in methods. They are dynamic; they do not retain their values across subsequent calls to the same method. Some *e* actions create implicit variables (see 2.3.3).

The scope of an *e* variable is the action block that encloses it. If a method contains nested action blocks, variables in the inner scopes hide the variables in the outer scopes. Variable scoping is described in more detail in 2.3.

The following sections describe the actions that create and modify *e* variables explicitly.

****Is this also correct for <+??**

16.2 var

Title	Variable declaration
Category	Action
Syntax	var <i>name</i> [: [<i>type</i>] [= <i>exp</i>]]
Parameters	<i>name</i> A legal <i>e</i> name.
	<i>type</i> A declared <i>e</i> type. The type can be omitted if the variable name is the same as the name of a struct type or if the variable is assigned a typed expression.
	<i>exp</i> The initial value of the variable. If no initial value is specified, the variables are initialized to 0 for integer types, NULL for structs, FALSE for Boolean types, and lists as empty.

This declares a new variable with the specified name as an element or list of elements of the specified type, and having an optional initial value.

The **var** action is legal in any place that an action is legal and the variable is recognized from that point on. The scope of an *e* variable is the action block that encloses it. If a method contains nested action blocks, variables in the inner scopes hide the variables in the outer scopes. Variable scoping is described in more detail in 2.3.

Syntax example:

```
var a: int;
```

16.3 =

Title	Simple assignment	
Category	Action	
Syntax	<i>lhs-exp=exp</i>	
Parameters	<i>lhs-exp</i>	A legal <i>e</i> expression that evaluates to a variable of a method, a global variable, a field of a struct, or an HDL object. The expression can contain the list index operator [<i>n</i>], the bit access operator [<i>i</i> : <i>j</i>], or the bit concatenation operator % { }.
	<i>exp</i>	A legal <i>e</i> expression, either an untyped expression (such as an HDL object) or an expression of the same type as the <i>lhs-exp</i> .

This assigns the value of the right-hand-side expression to the left-hand-side expression (see also Clause 3).

NOTE—There are two other places within the *e* language which make use of the equal sign. These are a double equal sign (==) for specifying equality in Boolean expression and a triple equal sign (===) for the Verilog-like identity operator. These two operators should not be confused with the single equal sign (=) assignment operator.

Syntax example:

```
sys.u = 0x2345;
```

16.4 op=

Title	Compound assignment	
Category	Action	
Syntax	<i>lhs-exp op=exp</i>	
Parameters	<i>lhs-exp</i>	A legal <i>e</i> expression that evaluates to a variable of a method, a global variable, a field of a struct, or an HDL object.
	<i>op</i>	A binary operator, including binary bitwise operators (except ~), the Boolean operators and and or , and the binary arithmetic operators.
	<i>exp</i>	A legal <i>e</i> expression of the same type as the <i>lhs-exp</i> .

This performs the specified operation on the two expressions and assigns the result to the left-hand-side expression.

Syntax example:

```
sys.c.count1 += 5;
```

16.5 <=

Title	Delayed assignment	
Category	Action	
Syntax	<i>[struct-exp.]field-name <= exp</i>	
Parameters	<i>struct-exp</i>	A legal <i>e</i> expression that evaluates to a struct. The default is me .
	<i>field-name</i>	A field of the struct referenced by <i>struct-exp</i> .
	<i>exp</i>	A legal <i>e</i> expression, either an untyped expression (such as an HDL object) or an expression of the same type as the <i>lhs-exp</i> .

The delayed assignment action assigns a struct field just before the next **@sys.new_time** after the action. The purpose is to support raceless coding in *e* by providing the same results regardless of the evaluation order of TCMs and temporal expressions. (See 8.4.3 for a description of **@sys.new_time**.) Both expressions are evaluated immediately (not delayed) in the current context. The assignment is not considered a time-consuming action, so you can use it in both TCMs and in regular methods, in **on** action blocks and in **exec** action blocks.

If a field has multiple delayed assignments in the same cycle, they are performed in the specified order. The final result is taken from the last delayed assignment action.

Unlike in HDL languages, the delayed assignment in *e* does not emit any events; thus, zero-delay iterations are not supported.

The left-hand-side expression in the delayed assignment action can only be a field. Unlike the assignment action, the delayed assignment action does not accept assignment to any of the following:

- A variable of a method
- A list item
- A bit
- A bit slice
- A bit concatenation expression

Example

The following example shows how delayed assignment provides raceless coding. In this example, there is one `incrementing()` TCM, which repeatedly increments the `sys.a` and `sys.da` fields, and one `observer()` TCM, which observes their value.

```
<'
extend sys {
  !a  : int;
  !da : int;
  incrementing()@any is {
    for i from 1 to 5 {
      a = a+1;
      da <= da+1;
      wait cycle;
    };
  stop_run();
};
```

```

1  |   observer()@any is {
      while (TRUE) {
          out( "observing 'a' as ", a, " observing 'da' as ", da );
          wait cycle;
5     };
      };

      run() is also {
10    start observer();
      start incrementing();
      };
15  };
    '>'

```

The following results show the value of `sys.a` observed by the `observer()` TCM is order-dependent, depending whether `observer()` is executed before or after `incrementing()`. The observed value of `sys.da`, however, is independent of the execution order. Even if `incrementing()` runs first, `sys.da` gets its incremented value just before the next **new_time** event and thus is not be seen by `observer()`.

If `observer()` runs before `incrementing()`:

```

      observing 'a' as 0 observing 'da' as 0
      -----
      observing 'a' as 1 observing 'da' as 1
      -----
25    observing 'a' as 2 observing 'da' as 2
      -----
      observing 'a' as 3 observing 'da' as 3
      -----
30    observing 'a' as 4 observing 'da' as 4
      -----

```

If `incrementing()` runs before `observer()`:

```

35    observing 'a' as 1 observing 'da' as 0
      -----
      observing 'a' as 2 observing 'da' as 1
      -----
      observing 'a' as 3 observing 'da' as 2
      -----
40    observing 'a' as 4 observing 'da' as 3
      -----
      observing 'a' as 5 observing 'da' as 4
      -----

```